

# Laboratory 2: Planning and Navigation

Drini Kerciku — 1004750780

Tian Yu — 1004750780

Ben Natra — 1005117366

Smit Patel — 1004734497

March 06, 2023

## 1 Introduction

An RRT and RRT\* subroutines are implemented in python to find optimal paths through the Willow garage maze and a custom map set up in our laboratory in Myhal. In the following sections, we briefly describe the modules used for planning in terms of collision detection, trajectory rollout and rewiring of nodes.

## 2 Report

### 2.1 Collision Detection

The **checkCollision(...)** function takes a series of way-points that are part of a trajectory to be tested for collision. Every location is mapped to the global reference frame based on the '.yaml' configuration file for the map and discretized through **point\_to\_cell(...)** called inside **points\_to\_robot\_circle(...)** function - a range of coordinates is generated to create an occupancy grid at the robot's location through the **disk(...)** method per way-point. Finally, each point in the occupancy grid of the robot at each location is used to check whether it is occupied by an obstacle or not, and the result is returned accordingly.

### 2.2 Simulate Trajectory

**simulate\_trajectory(...)** finds the trajectory that will nominally drive the robot from the start location to the end location nominally through the use of **robot\_controller(...)** and **trajectory\_rollout(...)** functions. Given the open ended problem for selecting the control inputs for the task, **robot\_controller(...)** searches for a feasible control pair within the defined time horizon through a grid of possible input combinations of linear and angular velocities - a pair that drives the agent the closest to the endpoint is selected with the aid of **trajectory\_rollout(...)** function calls. The trajectory of the returned inputs is simulated again and it is returned up to the point that is closest to the target based on our debug from run to run during development.

### 2.3 RRT Planning

The algorithm presented in class for RRT was closely followed in our implementation along with the structure provided in the skeleton code. A random point is sampled every iteration such that for the Willow garage map it is sampled within hardcoded bounds of the garage map to avoid generating paths in through the open space and around its exterior. Using the **cKDTree** object and its methods imported from the **scipy.spatial** module, we select the closest node in our tree to the sampled point and generate a nominal path towards it. It is important to note that we check for duplicate nodes with the endpoint of the returned trajectory, such that there are no points located within a 0.1 meter circular region about it. If there are no collisions along the trajectory, we add a new node to our tree and initialize its parents along with other needed information based on the requirements of the RRT planning algorithm.

## 2.4 Connect Node to Point

We are required to reconnect nodes for the RRT\* algorithm in the case of a better cost-to-come connection being available. A path is generated from the starting node to the end node depending on the relative position between the two locations and the robot's current heading. In the case when the nodes are too close or can be reached by driving the robot straight without changing its heading, the function **connect\_node\_to\_point(...)** returns with no results or a linearly interpolated path in 2D as per the logic implemented. Otherwise, a circular arc is generated constrained by the robot's heading, which serves as the tangent to the circle, and the two endpoints of the path to be defined. In order to make the implementation easier, the finish location is mapped into the robots reference frame through pose transformation to simplify the logic for interpolating the robot's heading along the path.

## 2.5 Trajectory Cost

The cost used in our implementation is that of Euclidean distance, such that for every two consecutive points along the trajectory, a distance is computed according to the distance metric and summed through for the entirety of the path.

## 2.6 Updating Children

A recursive approach is required for the updating the children of a node if rewiring is performed after finding the optimal connection for a node. We start with the parent node and recursively call **update\_children(...)** on its children nodes to update the cost-to-come up to the highest level of the tree (if one visualizes it hierarchically).

## 2.7 RRT\* Local Planning

The RRT\* follows the same procedure as the RRT algorithm presented in section 2.3 earlier in our report, with a few extra steps after adding the node to our tree. We find all the nearest neighbours with respect to the new node within a ball radius **rN** as a function of the current number of nodes in the tree, which is computed by through the **ball\_radius()** function, and search for a cost optimal connection through the list provided by the **cKDTree** methods - if a better connection is found in terms of cost, the newly added node information is updated. As the RRT\* algorithm dictates, optimal connections through the new node are queried and made if possible.

## 2.8 Local Planning

The functions for trajectory rollout and collision checking were reused, and adapted to support the path following algorithm along with the helper functions they depend on. Trajectories are propagated for a combination of input every iteration and checked for collision in order to filter them out of our options. In the **calculate\_cost** function, we prioritize controls that take us closer to the endpoint with as little difference as possible in input combinations from the previous control by assigning cost variables - we played around with the cost values and selected a combination that worked well for our paths and implementations. It is important to note that different tolerances were selected for Myhal and Willow garage simulations/demo to execute the path efficiently.

### 3 Paths

#### 3.1 Visuals

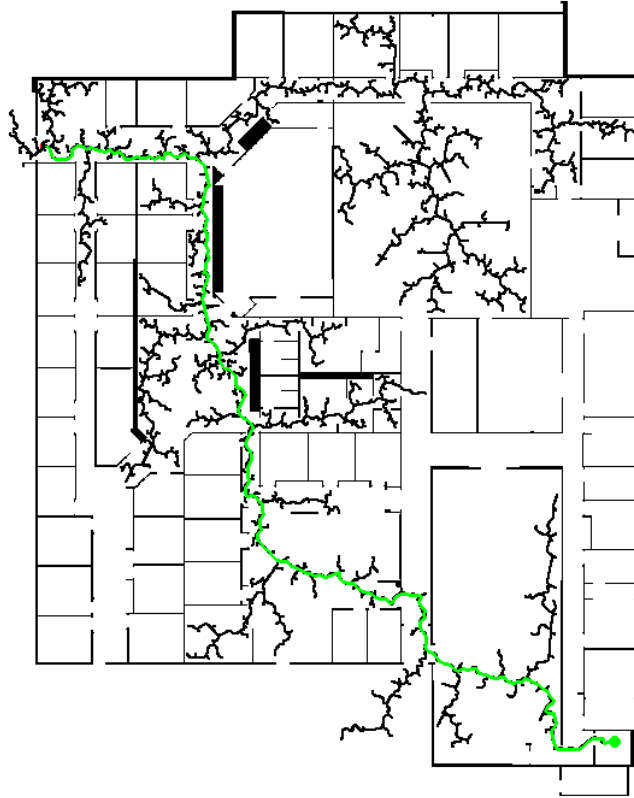


Figure 1: Tree and optimal path generated by RRT implementation.

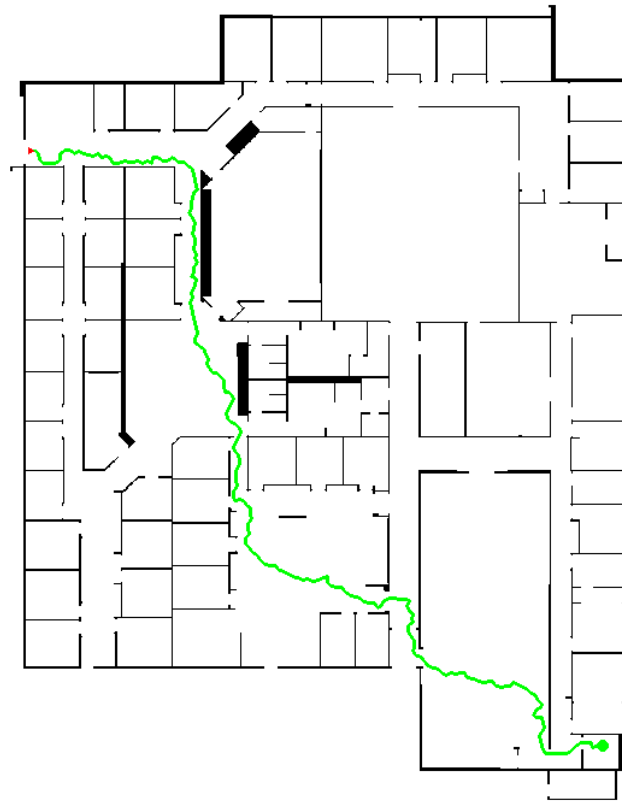


Figure 2: Optimal path generated by RRT implementation.



Figure 3: Node rewiring and tree generated by RRT\* implementation - red represents first order rewiring of the new nodes and blue indicates last node rewiring implemented in accordance with the skeleton code.

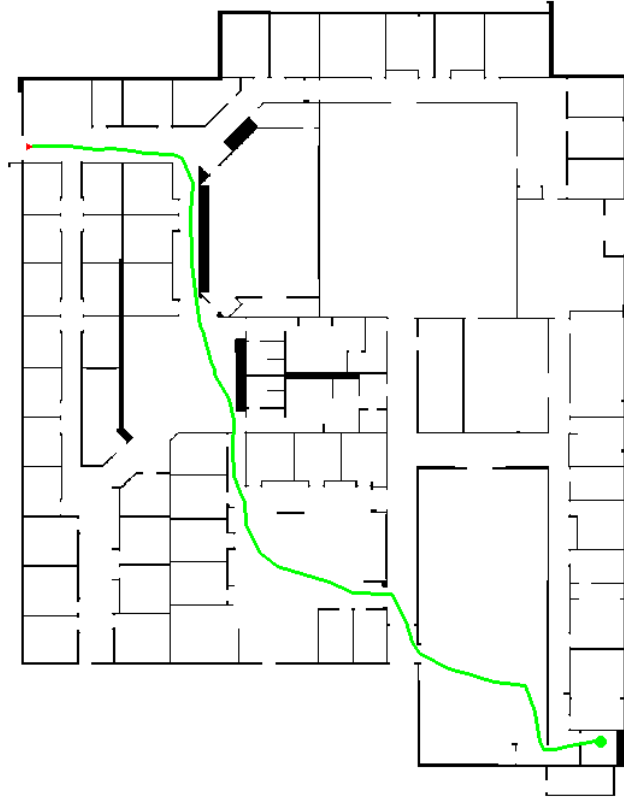


Figure 4: Optimal path generated by RRT\* implementation.

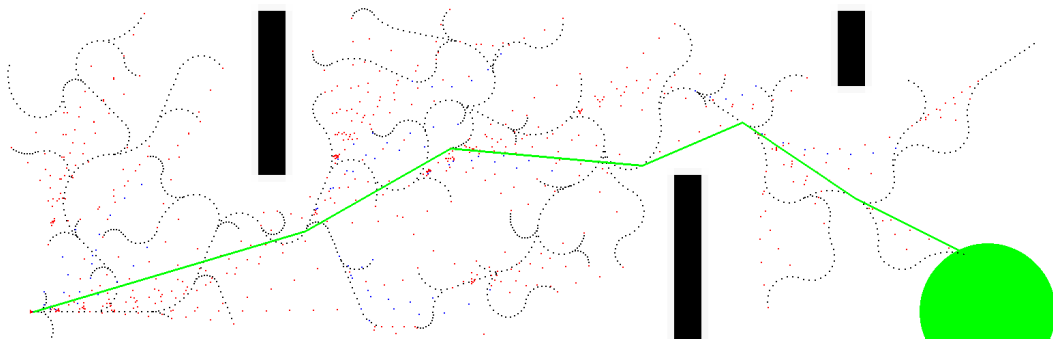


Figure 5: Optimal path generated by RRT\* implementation for the Myhal demo.

### 3.2 Comments

We were able to successfully solve the path planning problem and determine a seed that also provides the shortest path within the Willow garage map. Because of the rewiring using non-holonomic constraints, the trajectory generated by the RRT\* is much smoother and more optimal than the RRT path. It is also important to note that because of the rewiring and nature of rolling out trajectories with negative velocities, the orientation of the robot at certain locations can be flipped in the opposite direction of movement - in the future, one can post process the optimal path data to fix the orientation from one way-point to another such that the robot does not circle when reaching a location to match its orientation.

## 4 Simulation and Demo

A .zip file is included with our submission on Quercus containing the recordings of the Willow garage solution and in-person demo of solving the Myhal maze. We would like to note that VNC has not been working correctly for us in terms of resolution and the videos are a record of our playback in the lab. Moreover, we did not notice at the time that the location of goal was misplaced and the video shows the solution presented below for the Willow garage. Nevertheless, this demonstrates that our path following algorithm works successfully, and illustrates what was mentioned under Section 3.2 in terms of heading alignment of the robot. We noticed that when the following algorithm was ran in Turtlebot3, the robot seemed more stable and transitions from command to command were much smoother.

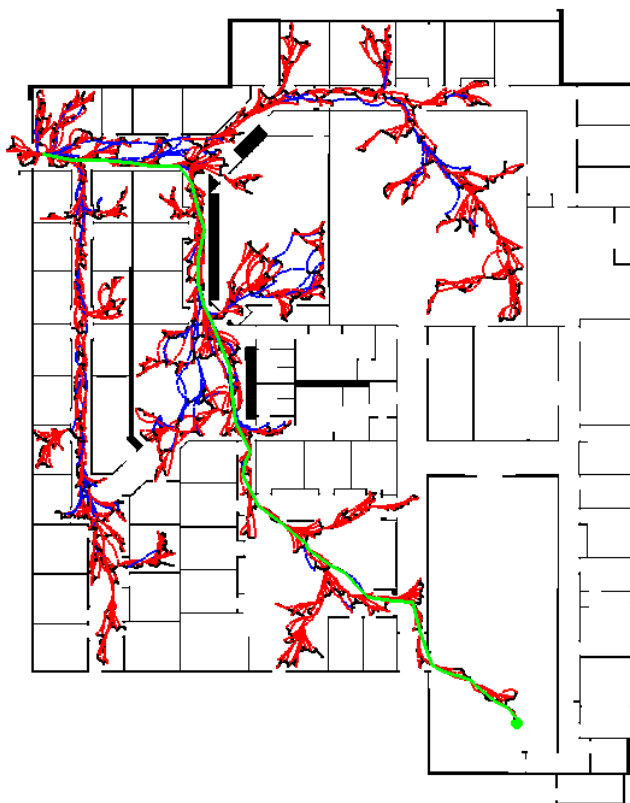


Figure 6: Node rewiring and tree generated by RRT\* implementation - red represents first order rewiring of the new nodes and blue indicates last node rewiring implemented in accordance with the skeleton code - path recorded in simulation.



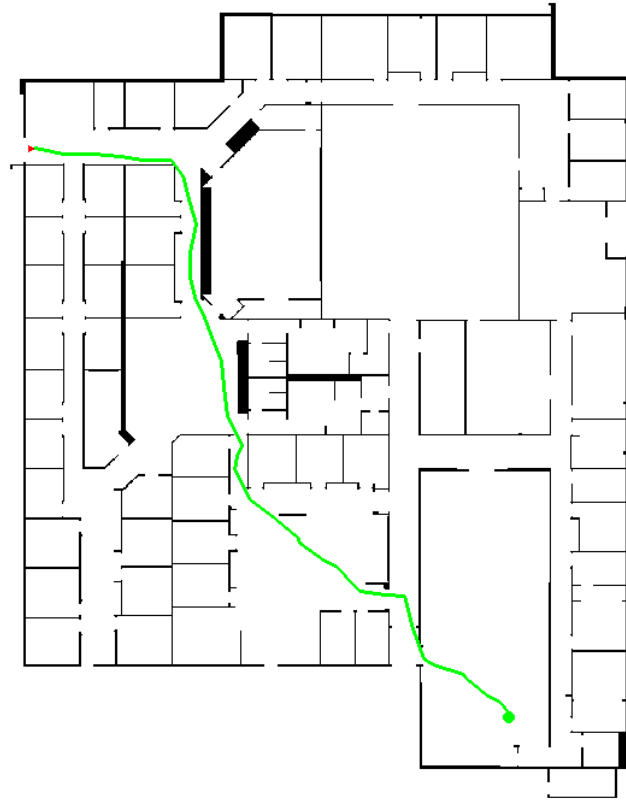


Figure 7: Optimal path generated by RRT\* implementation - path recorded in simulation.

## 5 Source Code

### 5.1 Planning : l2\_planning.py

```
1 #!/usr/bin/env python3
2  #Standard Libraries
3  import sys
4  import numpy as np
5  import yaml
6  import pygame
7  import time
8  import pygame_utils
9  import matplotlib.image as mpimg
10 from skimage.draw import disk
11 from scipy.linalg import block_diag
12 from scipy.spatial import cKDTree
13 import math
14 import matplotlib.pyplot as plt
15
16 ## Myhal SEED = 10 FOR RRT
17 ## Myhal SEED = 16 FOR RRT*
18 # RRT-STAR and RRT Seed - Willow (24)
19 np.random.seed(24)
20
21 def load_map(filename):
22     im = mpimg.imread("./maps/" + filename)
23     if len(im.shape) > 2:
24         im = im[:, :, 0]
25     im_np = np.array(im) #Whitespace is true, black is false
26     #im_np = np.logical_not(im_np)
27     return im_np
28
29 def load_map_yaml(filename):
30     with open("./maps/" + filename, "r") as stream:
31         map_settings_dict = yaml.safe_load(stream)
32     return map_settings_dict
33
34 #Node for building a graph
35 class Node:
36     def __init__(self, point, parent_id, cost):
37         self.point = point # A 3 by 1 vector [x, y, theta]
38         self.parent_id = parent_id # The parent node id that leads to this node (There
39         #should only every be one parent in RRT)
40         self.trajFromParent = None
41         self.cost = cost # The cost to come to this node
42         self.children_ids = [] # The children node ids of this node
43         return
44
45 #Path Planner
46 class PathPlanner:
47     #A path planner capable of performing RRT and RRT*
48     def __init__(self, map_filename, map_setings_filename, goal_point, stopping_dist):
49         #Get map information
50         self.occupancy_map = load_map(map_filename)
51         self.map_shape = self.occupancy_map.shape
52         self.map_settings_dict = load_map_yaml(map_setings_filename)
53
54         #Get the metric bounds of the map
55         self.bounds = np.zeros([2,2]) #m
56         self.bounds[0, 0] = self.map_settings_dict["origin"][0]
57         self.bounds[1, 0] = self.map_settings_dict["origin"][1]
58         self.bounds[0, 1] = self.map_settings_dict["origin"][0] + self.map_shape[1] *
59         self.map_settings_dict["resolution"]
60         self.bounds[1, 1] = self.map_settings_dict["origin"][1] + self.map_shape[0] *
61         self.map_settings_dict["resolution"]
62
63         # User Defined Variables
64         self.cellRes = self.map_settings_dict["resolution"]
65         self.originTheta = self.map_settings_dict["origin"][2]
66         self.originXY = np.array([[self.map_settings_dict["origin"][0]],
67                                     [self.map_settings_dict["origin"][1]]])
```

```

66     # map dimensions (bounds the region of the actual labyrinth not the whole 'world
67     ')
68     self.topL = [-3, 12]
69     self.topR = [45, -49]
70
71     #Robot information
72     self.robot_radius = 0.22 #m
73     ## CHOSEN TO MATCH THE follow_path.py range of inputs during trajectory rollout
74     self.vel_max = 0.28 #m/s (Feel free to change!)
75     self.rot_vel_max = 1.82 #rad/s (Feel free to change!)
76
77     self.v_options = np.linspace(-self.vel_max, self.vel_max, 5)
78     self.w_options = np.linspace(-self.rot_vel_max, self.rot_vel_max, 5)
79
80     self.mapIsMyhal = False
81
82     #Goal Parameters
83     self.goal_point = goal_point #m
84     self.stopping_dist = stopping_dist #m
85
86     #Trajectory Simulation Parameters
87     self.timestep = 1.5 #s
88     self.num_substeps = 10
89
90     #Planning storage
91     self.nodes = [Node(np.zeros((3,1)), -1, 0)]
92
93     #RRT* Specific Parameters
94     self.lebesgue_free = np.sum(self.occupancy_map) * self.map_settings_dict["
resolution"]**2
95     self.zeta_d = np.pi
96     self.gamma_RRT_star = 2*(1 + 1/2)**(1/2)*(self.lebesgue_free / self.zeta_d)
97     ** (1/2)
98     self.gamma_RRT = self.gamma_RRT_star + .1
99     self.epsilon = 2.5
100
101     ## FIX BUG WHEN LOADING THE MAP - pygame_utils.py was modified accordingly
102     # Pygame window for visualization
103     if map_filename == 'myhal.png':
104         shape = (self.occupancy_map.shape[1]*10, self.occupancy_map.shape[0]*10)
105         self.rot_vel_max = np.pi/2
106         self.v_options = np.linspace(0, self.vel_max, 5)
107         self.w_options = np.linspace(-self.rot_vel_max, self.rot_vel_max, 5)
108         self.num_substeps = 10
109         self.mapIsMyhal = True
110     else:
111         shape = (900, 900)
112
113     self.window = pygame_utils.PygameWindow(
114         "Path Planner", shape, self.occupancy_map.T.shape, self.map_settings_dict,
115         self.goal_point, self.stopping_dist, map_filename)
116     return
117
118 #Functions required for RRT
119 def sample_map_space(self):
120     #Return an [x,y] coordinate to drive the robot towards
121
122     probGoal = np.random.rand()
123
124     if probGoal < 0.05:
125         randX = self.goal_point[0][0] + 3*self.stopping_dist*np.random.rand()
126         randY = self.goal_point[1][0] + 3*self.stopping_dist*np.random.rand()
127         return np.array([[randX], [randY]])
128
129     if not self.mapIsMyhal:
130         randX = np.random.rand()*(self.topR[0] - self.topL[0]) + self.topL[0]
131         randY = np.random.rand()*(self.topR[1] - self.topL[1]) + self.topL[1]
132     else:
133         randX = np.random.rand()*(self.bounds[0, 1] - self.bounds[0, 0]) + self.
134         bounds[0, 0]
135         randY = np.random.rand()*(self.bounds[1, 1] - self.bounds[1, 0]) + self.
136         bounds[1, 0]

```

```

133         return np.array([[randX], [randY]])
134
135     def check_if_duplicate(self, point):
136         #Check if point is a duplicate of an already existing node
137
138         closest = self.closest_node(point)
139         closestPt = self.nodes[closest].point[:2,:].reshape((2,1))
140
141         if np.linalg.norm(point - closestPt) <= 0.1:
142             return True
143
144         return False
145
146     def closest_node(self, point):
147         #Returns the index of the closest node
148
149         dist = cKDTree(np.stack([node.point[:-1, :] for node in self.nodes], axis = 0).
squeeze(-1))
150         bestDist , Id = dist.query(point.T, k = 1)
151
152         return Id[0]
153
154     def simulate_trajectory(self, node_i: Node, point_s):
155         #Simulates the non-holonomic motion of the robot.
156         #This function drives the robot from node_i towards point_s. This function does
have many solutions!
157         #node_i is a 3 by 1 vector [x;y;theta] this can be used to construct the SE(2)
matrix T_{0I} in course notation
158         #point_s is the sampled point vector [x; y]
159
160         robot_traj = None
161
162         v, w = self.robot_controller(node_i.point, point_s)
163
164         trajBest = self.trajectory_rollout(v, w, node_i.point)
165
166         deltaXY = trajBest[:2,:] - point_s
167
168         batchDist = np.linalg.norm(deltaXY, axis = 0)
169
170         minId = np.argmin(batchDist)
171
172         robot_traj = trajBest[:,:(minId + 1)]
173
174         return robot_traj
175
176     def robot_controller(self, poseS, pointE):
177         #This controller determines the velocities that will nominally move the robot
from node i to node s
178         #Max velocities should be enforced
179
180         # initialize output to default values
181         v = 0
182         w = 0
183         # loop variable
184         bestDist = np.inf
185         # determine control law that takes us closest to end-point pointE
186         for i in range(0, len(self.v_options)):
187
188             v_test = self.v_options[i]
189
190             for j in range(0, len(self.w_options)):
191
192                 w_test = self.w_options[j]
193
194                 traj_ij = self.trajectory_rollout(v_test, w_test, poseS)
195
196                 deltaXY = traj_ij[:2,:] - pointE
197
198                 batchDist = np.linalg.norm(deltaXY, axis = 0)
199
200                 minId = np.argmin(batchDist)
201

```

```

202         if batchDist[minId] < bestDist:
203             v = v_test
204             w = w_test
205             bestDist = batchDist[minId]
206
207         return v, w
208
209     def trajectory_rollout(self, vel, rot_vel, x_0):
210
211         # Given your chosen velocities determine the trajectory of the robot for your
212         # given timestep
213         # The returned trajectory should be a series of points to check for collisions
214
215         #  $x_{k+1} = x_k + B(x_k)u_k \Delta t$ 
216         B = lambda x_theta : np.array([[np.cos(x_theta), 0], [np.sin(x_theta), 0], [0,
217 1]])
218
219         # initialize output
220         trajectory = x_0
221
222         # prepare variables needed for computation
223         dt = self.timestep/self.num_substeps
224         u = np.array([[vel], [rot_vel]])
225         # control is computed from the current pose
226         # in the global reference frame
227         currState = x_0
228         nextState = np.zeros((3,1))
229
230         for i in range(0, self.num_substeps):
231
232             # compute  $B(x_k)$ 
233             B_k = B(currState[2][0])
234             # compute new state
235             nextState = currState + np.dot(B_k, u)*dt
236
237             # adjust heading to be in  $[-\pi, \pi]$ 
238             if nextState[2] > np.pi:
239                 nextState[2] = nextState[2] - 2*np.pi
240             elif nextState[2] < -np.pi:
241                 nextState[2] = nextState[2] + 2*np.pi
242
243             # save current state and progress to the next timestep
244             trajectory = np.hstack((trajectory, nextState))
245
246             currState = nextState
247
248         return trajectory
249
250     def point_to_cell(self, point):
251
252         #Convert a series of [x,y] points in the map to the indices for the
253         #corresponding cell in the occupancy map
254         #point is a 2 by N matrix of points of interest
255
256         # compute rotation matrix to map points on the reference
257         # frame if a rotation is present
258         theta = self.originTheta
259         R = np.array([[np.cos(theta), np.sin(theta)],
260 [-np.sin(theta), np.cos(theta)]])
261
262         # transform points to cell coordinates
263         pts_origin = (np.dot(R, point) - np.dot(R, self.originXY))/self.cellRes
264
265         ptsX = pts_origin[0, :]
266         ptsY = self.map_shape[0] - pts_origin[1, :]
267         ptsOut = np.vstack((ptsX, ptsY))
268
269         return ptsOut.astype(int) #(2, N)
270
271     def points_to_robot_circle(self, points):
272
273         #Convert a series of [x,y] points to robot map footprints for collision
274         #detection

```

```

271     #Hint: The disk function is included to help you with this function
272
273     # transform robot coordinates to cell coordinates
274     cellPos = self.point_to_cell(points)
275
276     # create a base region for the radius specified
277     radRes = int(self.robot_radius/self.cellRes) + 1
278
279     # attempt to be more conservative with Myhal map
280     if self.mapIsMyhal:
281         radRes += 1
282
283     rr, cc = disk((0, 0), radRes)
284     baseCoords = np.vstack((rr,cc), dtype = np.int32)
285
286     # create result place holder
287     tempX = np.zeros((2,1))
288     pts2Rob = {}
289
290     for i in range(0, cellPos.shape[1]):
291
292         # check for duplicates
293         keyLoc = (cellPos[0][i], cellPos[1][i])
294
295         if keyLoc not in pts2Rob:
296
297             # extract current points
298             tempX[0] = cellPos[0][i]
299             tempX[1] = cellPos[1][i]
300             # map the occupancy region
301             tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
302             # adjust for corner cases
303             tempDiff[0, tempDiff[0] >= self.occupancy_map.shape[1]] = self.
occupancy_map.shape[1] - 1
304             tempDiff[1, tempDiff[1] >= self.occupancy_map.shape[0]] = self.
occupancy_map.shape[0] - 1
305
306             remCells = np.where(tempDiff < 0)[1]
307             tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
308             # save in dictionary only unique coordinates and entries
309             pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)
310
311     return pts2Rob
312
313 def checkCollision(self, points):
314
315     # points are of shape (2, N) in discrete coordinates
316
317     # get the occupancy map for each region
318     pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy
regions
319
320     # check if pose is collision free
321     for key in pts2RobPoses:
322
323         # get occupancy map: (2, M) array
324         mapOcc = pts2RobPoses[key].astype(int)
325
326         # check if any of the cells contains a value of 0 and report collision as
True
327         crash = np.any(self.occupancy_map[mapOcc[1,:], mapOcc[0, :]] == 0)
328
329         if crash:
330             return True
331
332     return False
333
334 #Note: If you have correctly completed all previous functions, then you should be
able to create a working RRT function
335
336 #RRT* specific functions
337 def ball_radius(self):
338     #Close neighbor distance

```

```

339         card_V = len(self.nodes)
340         return min(self.gamma_RRT * (np.log(card_V) / card_V ) ** (1.0/2.0), self.
epsilon)
341
342     def connect_node_to_point(self, node_i, point_f):
343         #Given two nodes find the non-holonomic path that connects them
344         #Settings
345         #node is a 3 by 1 node
346         #point is a 2 by 1 point
347
348         # initialize return variable
349         resTraj = None
350         tooClose = False
351
352         # partition input node
353         point_s = node_i[:2,0].reshape(2,1)
354         robHead = node_i[2,0]
355
356         # important info : distance and bearing angle to the robots reference frame
357         dist = np.linalg.norm(point_s - point_f)
358         theta = np.arctan2(point_f[1,0] - point_s[1,0], point_f[0,0] - point_s[0,0])
359
360         if dist <= 0.1:
361
362             tooClose = True
363
364         elif (abs(theta - robHead) < 1e-3) or ((abs(abs(theta - robHead) - np.pi) < 1e
-3)):
365
366             # choose an appropriate step resolution according to the distance
367             stepNum = 10
368             resTraj = np.linspace(point_s, point_f, num = stepNum)[:,:,:0].T
369             resTraj = np.vstack((resTraj, robHead*np.ones((1, stepNum))))
370
371         else:
372
373             # tranform point_f to robots frame through T_vw
374             R_vw = np.array([[np.cos(robHead), np.sin(robHead)],
375                             [-np.sin(robHead), np.cos(robHead)]])
376             o_vw = - np.dot(R_vw, point_s)
377
378             T_vw = np.hstack((R_vw, o_vw))
379             T_vw = np.vstack((T_vw, np.array([[0,0,1]])))
380             T_inv = np.linalg.inv(T_vw)
381
382             # augment point_f
383             point_f_w = np.vstack((point_f, np.array([[1]])))
384             point_f_v = np.dot(T_inv, point_f_w)
385
386             # compute the the center of circle by using vector S->F
387             alpha = point_f_v[0,0]/point_f_v[1,0]
388             xC = 0
389             yC = alpha*point_f_v[0,0]/2 + point_f_v[1,0]/2
390             # radius of circle
391             r = abs(yC)
392
393             # compute arc-angle
394             cosRho = 1 - 2*((dist/(2*r))**2)
395
396             # ILL-CONDITIONED PROBLEM - SKIP CONNECTION
397             if abs(abs(cosRho) - 1) < 1e-6:
398                 return (True, None)
399
400             rho = np.arctan2(np.sqrt(1-cosRho**2), cosRho)
401
402             # compute arc-length
403             l = rho*r
404             # select an appropriate stepsize
405             stepNum = np.ceil(l/self.robot_radius).astype(np.int32) + 1
406             # determine direction of traversing the arc length for
407             # each of the possible combinations (4 to consider)
408
409             dirY = -1 # decreasing with each substep

```

```

410         dirX = -1 # decreasing with each substep
411
412         if yC > 0:
413             dirY = 1
414
415         if point_f_v[0,0] > 0:
416             dirX = 1
417
418         # start determining the trajectory
419         dRho = rho/stepNum
420         resTraj = node_i
421         rhoAcc = 0
422
423         ## FOR DEBUGGING
424         # plt.figure()
425         # plt.plot(0, 0, 'rX')
426         # plt.plot(point_f_v[0,0], point_f_v[1,0], 'ro')
427
428         for i in range(0, stepNum + 1):
429
430             # compute next waypoint in vehicle frame
431             currPt = np.array([[r*np.sin(rhoAcc)*dirX + xC],
432                               [-r*np.cos(rhoAcc)*dirY + yC],
433                               [1]])
434
435             ## FOR DEBUGGING
436             # plt.plot(currPt[0,0], currPt[1,0], 'b.')

```



```

481     parent = self.nodes[node_id]
482
483     # iterate through the children recursively and update the cost
484     for childID in parent.children_ids:
485         trajParent = self.nodes[childID].trajFromParent
486         self.nodes[childID].cost = parent.cost + self.cost_to_come(trajParent)
487         self.update_children(childID)
488
489     return
490
491 #Planner Functions
492 def rrt_planning(self):
493     #This function performs RRT on the given map and robot
494     #You do not need to demonstrate this function to the TAs, but it is left in for
    you to check your work
495
496     i = 0
497
498     while True: #Most likely need more iterations than this to complete the map!
499
500         #Sample map space
501         point = self.sample_map_space()
502
503         #Get the closest point
504         closest_node_id = self.closest_node(point)
505
506         # Simulate driving the robot towards the closest point
507         trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id], point)
508
509         # check how close we were able to reach our goal (we may not be able
510         # to reach the point exactly or at all)
511         lastPt = trajectory_o[:, -1].reshape((3,1))
512         lastPt2D = lastPt[:2,0].reshape((2,1))
513
514         # check if a similar point is already in our tree
515         checkDuplicate = self.check_if_duplicate(lastPt2D)
516
517         if checkDuplicate:
518             continue
519
520         didCollide = self.checkCollision(trajectory_o[:2,:])
521
522         if not didCollide:
523
524             # add new node on our list
525             newNode = Node(lastPt, closest_node_id, 0)
526             newNode.trajFromParent = trajectory_o
527             self.nodes.append(newNode)
528             self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)
529
530             # check if we are at goal
531             if np.linalg.norm(lastPt2D - self.goal_point) < self.stopping_dist:
532                 break
533
534             # # FOR DEBUGGING
535             # for j in range(0, trajectory_o.shape[1]):
536             #     self.window.add_point(trajectory_o[:,-1, j].copy())
537
538         i += 1
539
540     return self.nodes
541
542 def rrt_star_planning(self):
543     #This function performs RRT* for the given map and robot
544     while True: #Most likely need more iterations than this to complete the map!
545         #Sample map space
546         point = self.sample_map_space()
547
548         #Get the closest point
549         closest_node_id = self.closest_node(point)
550
551         # Simulate driving the robot towards the closest point
552         trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id], point)

```

```

553
554     # check how close we were able to reach our goal (we may not be able
555     # to reach the point exactly or at all)
556     lastPt = trajectory_o[:, -1].reshape((3,1))
557     lastPt2D = lastPt[:2,0].reshape((2,1))
558
559     # check if a similar point is already in our tree
560     checkDuplicate = self.check_if_duplicate(lastPt2D)
561
562     if checkDuplicate:
563         continue
564
565     didCollide = self.checkCollision(trajectory_o[:2,:])
566
567     if not didCollide:
568
569         # add new node on our list
570         newNode = Node(lastPt, closest_node_id, 0)
571         newNode.trajFromParent = trajectory_o
572         newNode.cost = self.cost_to_come(trajectory_o) + self.nodes[
closest_node_id].cost
573         self.nodes.append(newNode)
574         self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)
575
576         # compute radius of neighbours
577         rN = self.ball_radius()
578
579         # find nearest neighbours within rN
580         dist = cKDTree(np.stack([node.point[:-1, :] for node in self.nodes],
axis = 0).squeeze(-1))
581         Id = dist.query_ball_point(lastPt[:2,0], r = rN)
582
583         currCost = newNode.cost
584         newId = len(self.nodes) - 1
585         currParentId = closest_node_id
586         currTraj = trajectory_o
587
588         ## FOR DEBUGGING
589         # for j in range(0, trajectory_o.shape[1]):
590         #     self.window.add_point(trajectory_o[:-1, j].copy(), color = (0,0,0)
591     )
592
593     lowerCost = False
594
595     #Last node rewire
596     for n in Id:
597
598         closeFlag, tempTraj = self.connect_node_to_point(self.nodes[n].point
, lastPt2D)
599
600         if (n == newId) or closeFlag:
601             continue
602
603         # check for collision
604         didCollideTemp = self.checkCollision(tempTraj[:2,:])
605
606         if not didCollideTemp:
607
608             # compute cost
609             tempCost = self.cost_to_come(tempTraj) + self.nodes[n].cost
610
611             if currCost >= tempCost:
612                 currCost = tempCost
613                 currTraj = tempTraj
614                 currParentId = n
615                 #lowerCost = True
616
617         # rewire child
618         if currParentId != closest_node_id or lowerCost:
619             # remove child from the list
620             self.nodes[closest_node_id].children_ids.remove(newId)
621             # update new parent data

```

```

622         self.nodes[newId].trajFromParent = currTraj
623         self.nodes[newId].cost = currCost
624         self.nodes[newId].parent_id = currParentId
625         self.nodes[currParentId].children_ids.append(newId)
626
627         ## FOR DEBUGGING
628         # for j in range(0, currTraj.shape[1]):
629         #     self.window.add_point(currTraj[:-1, j].copy(), color =
(255,0,0))
630
631         # Close node rewire
632         for n in Id:
633
634             # temporary variable
635             tempPid = self.nodes[n].parent_id
636
637             closeFlag, tempTraj = self.connect_node_to_point(lastPt, self.nodes[
n].point[:2,0].reshape((2,1)))
638
639             if closeFlag or (n == newId):
640                 continue
641
642             # check for collision
643             didCollideTemp = self.checkCollision(tempTraj[:2,:])
644
645             if not didCollideTemp:
646
647                 # compute cost and rewire if possible
648                 tempCost = self.cost_to_come(tempTraj) + self.nodes[newId].cost
649
650                 if tempCost <= self.nodes[n].cost:
651                     self.nodes[tempPid].children_ids.remove(n)
652                     self.nodes[n].parent_id = newId
653                     self.nodes[n].cost = tempCost
654                     self.nodes[n].trajFromParent = tempTraj
655                     self.nodes[newId].children_ids.append(n)
656                     self.update_children(n)
657
658                     ## FOR DEBUGGING
659                     # for j in range(0, currTraj.shape[1]):
660                     #     self.window.add_point(currTraj[:-1, j].copy(), color =
(0,0,255))
661
662
663             # check if we are at goal
664             if np.linalg.norm(lastPt2D - self.goal_point) < self.stopping_dist:
665                 break
666
667         return self.nodes
668
669     def recover_path(self, node_id = -1):
670         path = [self.nodes[node_id].point]
671         current_node_id = self.nodes[node_id].parent_id
672         while current_node_id > -1:
673             path.append(self.nodes[current_node_id].point)
674             current_node_id = self.nodes[current_node_id].parent_id
675         path.reverse()
676         return path
677
678     def main():
679
680         #Set map information
681         # map_filename = "myhal.png"
682         # map_setings_filename = "myhal.yaml"
683         # #robot information
684         # goal_point = np.array([[7], [0]]) #m
685
686         #Set map information
687         map_filename = "willowgarageworld_05res.png"
688         map_setings_filename = "willowgarageworld_05res.yaml"
689
690         #robot information
691         goal_point = np.array([[42], [-44]]) #m

```

```

692     stopping_dist = 0.5 #m
693
694
695     # HAD SOME ISSUES IN MY VM SO I INCREASED THE LIMIT
696     sys.setrecursionlimit(4000)
697     print("Recursion limit set to {}".format(sys.getrecursionlimit()))
698
699     #RRT precursor
700     path_planner = PathPlanner(map_filename, map_settings_filename, goal_point,
701     stopping_dist)
702     nodes = path_planner.rrt_planning()
703     node_path_metric = np.hstack(path_planner.recover_path())
704
705     for i in range (1, node_path_metric.shape[1]):
706         pt1 = node_path_metric[:, i-1].reshape((3,1))
707         pt2 = node_path_metric[:, i].reshape((3,1))
708         path_planner.window.add_line(pt1[:2, 0].copy(), pt2[:2, 0].copy(), width = 3,
709         color = (0, 255, 0))
710
711     pygame.image.save(path_planner.window.screen, f"image.png")
712
713     #Leftover test functions
714     np.save("shortest_path.npy", node_path_metric)
715
716 if __name__ == '__main__':
717     main()

```

## 5.2 Navigation : l2\_follow\_path.py

```

1  #!/usr/bin/env python3
2  from __future__ import division, print_function
3  import os
4
5  import numpy as np
6  from scipy.linalg import block_diag
7  from skimage.draw import disk
8  from scipy.spatial.distance import cityblock
9  import rospy
10 import tf2_ros
11 import matplotlib.pyplot as plt
12
13 # msgs
14 from geometry_msgs.msg import TransformStamped, Twist, PoseStamped
15 from nav_msgs.msg import Path, Odometry, OccupancyGrid
16 from visualization_msgs.msg import Marker
17
18 # ros and se2 conversion utils
19 import utils
20
21
22 TRANS_GOAL_TOL = .25 # m, tolerance to consider a goal complete
23 ROT_GOAL_TOL = .6 #.6 # rad, tolerance to consider a goal complete
24 TRANS_VEL_OPTS = [0, 0.025, 0.13, 0.26] # m/s, max of real robot is .26
25 ROT_VEL_OPTS = np.linspace(-1.82, 1.82, 11) # rad/s, max of real robot is 1.82
26 CONTROL_RATE = 5 # Hz, how frequently control signals are sent
27 CONTROL_HORIZON = 5 # seconds. if this is set too high and INTEGRATION_DT is too low,
28     code will take a long time to run!
29 # INTEGRATION_DT = .025 # s, delta t to propagate trajectories forward by
30 INTEGRATION_DT = .05
31 COLLISION_RADIUS = 0.225 # m, radius from base_link to use for collisions, min of
32     0.2077 based on dimensions of .281 x .306
33 ROT_DIST_MULT = .1 # multiplier to change effect of rotational distance in choosing
34     correct control
35 OBS_DIST_MULT = .1 # multiplier to change the effect of low distance to obstacles on a
36     path
37 MIN_TRANS_DIST_TO_USE_ROT = TRANS_GOAL_TOL # m, robot has to be within this distance to
38     use rot distance in cost
39 PATH_NAME = 'path.npy' # saved path from l2_planning.py, should be in the same
40     directory as this file
41
42 # here are some hardcoded paths to use if you want to develop l2_planning and this file
43     in parallel
44 # TEMP_HARDCODE_PATH = [[2, 0, 0], [2.75, -1, -np.pi/2], [2.75, -4, -np.pi/2], [2, -4.4,

```

```

    np.pi]] # almost collision-free
38 TEMP_HARDCODE_PATH = [[2, -.5, 0], [2.4, -1, -np.pi/2], [2.45, -3.5, -np.pi/2], [1.5,
    -4.4, np.pi]] # some possible collisions
39
40
41 class PathFollower():
42     def __init__(self):
43         # time full path
44         self.path_follow_start_time = rospy.Time.now()
45
46         # use tf2 buffer to access transforms between existing frames in tf tree
47         self.tf_buffer = tf2_ros.Buffer()
48         self.listener = tf2_ros.TransformListener(self.tf_buffer)
49         rospy.sleep(1.0) # time to get buffer running
50
51         # constant transforms
52         self.map_odom_tf = self.tf_buffer.lookup_transform('map', 'odom', rospy.Time(0),
    rospy.Duration(2.0)).transform
53         # print("map odom tf:")
54         # print(self.map_odom_tf)
55
56         # subscribers and publishers
57         self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
58         self.global_path_pub = rospy.Publisher('~global_path', Path, queue_size=1, latch
    =True)
59         self.local_path_pub = rospy.Publisher('~local_path', Path, queue_size=1)
60         self.collision_marker_pub = rospy.Publisher('~collision_marker', Marker,
    queue_size=1)
61
62         # map
63         map = rospy.wait_for_message('/map', OccupancyGrid)
64         self.map_np = np.array(map.data).reshape(map.info.height, map.info.width)
65         self.map_resolution = round(map.info.resolution, 5)
66         self.map_origin = -utils.se2_pose_from_pose(map.info.origin) # negative because
    of weird way origin is stored
67         # print("map origin and map")
68         # print(self.map_origin)
69         self.map_nonzero_idxes = np.argwhere(self.map_np) #prob use this for colision
    detection
70         # print(map.info)
71         # print(np.shape(self.map_np))
72
73         # for debug
74         self.map_xy = np.nonzero(self.map_np)
75         # print(map_xy)
76         # plt.scatter(map_xy[0], map_xy[1])
77         # plt.show()
78         # exit()
79
80
81         # collisions
82         self.collision_radius_pix = COLLISION_RADIUS / self.map_resolution
83         self.collision_marker = Marker()
84         self.collision_marker.header.frame_id = '/map'
85         self.collision_marker.ns = '/collision_radius'
86         self.collision_marker.id = 0
87         self.collision_marker.type = Marker.CYLINDER
88         self.collision_marker.action = Marker.ADD
89         self.collision_marker.scale.x = COLLISION_RADIUS * 2
90         self.collision_marker.scale.y = COLLISION_RADIUS * 2
91         self.collision_marker.scale.z = 1.0
92         self.collision_marker.color.g = 1.0
93         self.collision_marker.color.a = 0.5
94
95         # transforms
96         self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link', rospy
    .Time(0), rospy.Duration(2.0))
97         self.pose_in_map_np = np.zeros(3)
98         self.pos_in_map_pix = np.zeros(2)
99         self.update_pose()
100
101         # path variables
102         cur_dir = os.path.dirname(os.path.realpath(__file__))

```

```

103
104     # to use the temp hardcoded paths above, switch the comment on the following two
105     lines
106     self.path_tuples = np.load(os.path.join(cur_dir, 'RRT_star_willow_seed_7.npy')).
107     T
108     #self.path_tuples = np.array(TEMP_HARDCODE_PATH)
109
110     self.path = utils.se2_pose_list_to_path(self.path_tuples, 'map')
111     self.global_path_pub.publish(self.path)
112
113     # goal
114     self.cur_goal = np.array(self.path_tuples[0])
115     self.cur_path_index = 0
116
117     # trajectory rollout tools
118     # self.all_opts is a Nx2 array with all N possible combinations of the t and v
119     vels, scaled by integration dt
120     self.all_opts = np.array(np.meshgrid(TRANS_VEL_OPTS, ROT_VEL_OPTS)).T.reshape
121     (-1, 2)
122
123     # if there is a [0, 0] option, remove it
124     all_zeros_index = (np.abs(self.all_opts) < [0.001, 0.001]).all(axis=1).nonzero()
125     [0]
126     if all_zeros_index.size > 0:
127         self.all_opts = np.delete(self.all_opts, all_zeros_index, axis=0)
128     self.all_opts_scaled = self.all_opts * INTEGRATION_DT
129
130     self.num_opts = self.all_opts_scaled.shape[0]
131     self.horizon_timesteps = int(np.ceil(CONTROL_HORIZON / INTEGRATION_DT))
132
133     self.rate = rospy.Rate(CONTROL_RATE)
134     self.prev_ctrl = np.array([0,0])
135
136     rospy.on_shutdown(self.stop_robot_on_shutdown)
137     self.follow_path()
138
139     def trajectory_rollout(self, vel, rot_vel, x_0):
140
141         # Given your chosen velocities determine the trajectory of the robot for your
142         given timestep
143         # The returned trajectory should be a series of points to check for collisions
144
145         # x_0 is expected as x, y, theta
146         # x_k_1 = x_k + B(x_k)u_k*dt
147         B = lambda x_theta : np.array([[np.cos(x_theta), 0],[np.sin(x_theta), 0],[0,
148         1]])
149
150         # initialize output
151         trajectory = np.zeros((3, 1))
152
153         # prepare variables needed for computation
154         u = np.array([[vel],[rot_vel]])
155         dt = INTEGRATION_DT
156         # control is computed from the current pose
157         # in the global reference frame
158         currState = np.reshape(x_0, (3,1))
159         nextState = np.zeros((3,1))
160
161         for i in range(0, self.horizon_timesteps):
162
163             # compute B(x_k)
164             B_k = B(currState[2][0])
165             # compute new state
166             #print(np.shape(B_k))
167             #print(np.shape(u))
168             nextState = currState + np.dot(B_k,u)*dt
169
170             # print("state propagation")
171             # print(currState)
172             # print(nextState)
173
174             # adjust heading to be in [-pi, pi]
175             if nextState[2] > np.pi:

```

```

169         nextState[2] = nextState[2] - 2*np.pi
170     elif nextState[2] < -np.pi:
171         nextState[2] = nextState[2] + 2*np.pi
172
173     # save current state and progress to the next timestep
174     if i == 0:
175         trajectory = nextState
176     else:
177         trajectory = np.hstack((trajectory, nextState))
178
179     currState = nextState
180
181     return np.transpose(trajectory)
182
183 def points_to_robot_circle(self, points):
184     #Convert a series of [x,y] points to robot map footprints for collision
185     detection
186     #Hint: The disk function is included to help you with this function
187
188     # transform robot coordinates to cell coordinates
189     # cellPos = self.point_to_cell(points)
190     cellPos = points
191
192     # create a base region for the radius specified
193     radRes = int(COLLISION_RADIUS/self.map_resolution) # TODO: check if self map
194     resolution is legit
195     rr, cc = disk((0, 0), radRes)
196     baseCoords = np.vstack((rr,cc), dtype = np.int32)
197
198     # create result place holder
199     tempX = np.zeros((2,1))
200     pts2Rob = {}
201
202     for i in range(0, cellPos.shape[1]):
203
204         # check for duplicates
205         keyLoc = (cellPos[0][i], cellPos[1][i])
206
207         if keyLoc not in pts2Rob:
208
209             # extract current points
210             tempX[0] = cellPos[0][i]
211             tempX[1] = cellPos[1][i]
212             # map the occupancy region
213             tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
214             # adjust for corner cases
215             tempDiff[0, tempDiff[0] >= self.map_np.shape[1]] = self.map_np.shape[1]
216             tempDiff[1, tempDiff[1] >= self.map_np.shape[0]] = self.map_np.shape[0]
217
218             remCells = np.where(tempDiff < 0)[1]
219             tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
220             # save in dictionary only unique coordinates and entries
221             pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)
222
223     return pts2Rob
224
225 def checkCollision(self, points_T):
226     # points are of shape (2, N) in discrete coordinates
227     points = np.transpose(points_T)
228     # get the robot occupancy map for each set of center points
229     pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy
230     regions
231     # dictionary with keys being center points, and data being an array of
232     coordinates of rob occupancy
233
234     # check if pose is collision free
235     for key in pts2RobPoses:
236
237         # get robot occupancy map: (2, M) array
238         mapOcc = pts2RobPoses[key].astype(int) # coordiantes based on set passed in
239         by "points"
240         # print("occupied cells by key:{}".format(key))

```

```

235         # print(mapOcc)
236
237         # plt.scatter(-self.map_xy[0], self.map_xy[1])
238         # plt.scatter(-mapOcc[1,:], mapOcc[0,:])
239         # plt.scatter(-key[1], key[0], s = 100)
240         # plt.scatter(-points[1,0], points[0,0])
241         # plt.show()
242         # exit()
243
244         # check if any of the environment occupancy map cells occupied by the robot
contains a value of 0 and report collision as True
245         crash = np.any(self.map_np[mapOcc[1, :], mapOcc[0, :]] == 100) # TODO:
ensure this is how map_np works
246         if crash:
247             return True
248
249         return False
250
251     def calculate_cost(self, cand_opt, end_pt):
252         # 2 preferences: low dist from goal, low change from previous ctrl
253         trans_factor = 16
254         rot_factor = 0
255         diff_rot_factor = 0
256
257         # difference in goal poses penalized
258         curr_goal = self.cur_goal
259         pose_diff = np.abs(curr_goal - end_pt)
260         loc_cost = trans_factor*(pose_diff[0] + pose_diff[1])
261         rot_cost = rot_factor*pose_diff[2] / loc_cost
262
263         # change in control penalize
264         curr_trans_opt = cand_opt[0]
265         last_trans_opt = self.prev_ctrl[0]
266         curr_rot_opt = cand_opt[1]
267         last_rot_opt = self.prev_ctrl[1]
268         ctrl_chg_cost = np.abs(curr_trans_opt - last_trans_opt) + diff_rot_factor*np.abs
(curr_rot_opt - last_rot_opt)
269
270         return loc_cost + rot_cost + ctrl_chg_cost # TODO: maybe implement a saturating
cost for rotational error
271
272     def follow_path(self):
273         while not rospy.is_shutdown():
274             # timing for debugging...loop time should be less than 1/CONTROL_RATE
275             tic = rospy.Time.now()
276
277             self.update_pose()
278             self.check_and_update_goal()
279
280             t1 = rospy.Time.now()
281             # start trajectory rollout algorithm
282             local_paths = np.zeros([self.horizon_timesteps + 1, self.num_opts, 3])
283             local_paths[0] = np.atleast_2d(self.pose_in_map_np).repeat(self.num_opts,
axis=0)
284
285             # propagate trajectory forward, assuming perfect control of velocity and no
dynamic effects
286
287             for i, opts in enumerate(self.all_opts):
288                 trans_vel = opts[0]
289                 rot_vel = opts[1]
290                 pred_traj = self.trajectory_rollout(trans_vel, rot_vel, self.
pose_in_map_np)
291                 local_paths[1:,i,:] = pred_traj #assign complete trajectory one at a
time
292                 # print(local_paths[:,0,:])
293                 # print("start")
294                 # print(local_paths[0,:,:])
295                 # print("end pts")
296                 # print(local_paths[-1,:,:])
297
298                 t2 = rospy.Time.now()
299                 traj_rollout_time = t2 - t1

```



```

300
301     # check all trajectory points for collisions
302     # first find the closest collision point in the map to each local path point
303     local_paths_pixels = np rint((self.map_origin[:2] + local_paths[:, :, :2]) /
self.map_resolution).astype(int) #round to nearest int as index
304     # print("local path shape")
305     # print(np.shape(local_paths_pixels))
306     # plt.scatter(-self.map_xy[0], self.map_xy[1])
307     # plt.scatter(-local_paths_pixels[0,:,1], local_paths_pixels[0,:,0])
308     # print(-local_paths_pixels[0,0,0:2])
309     #
310     # print(-local_paths_pixels[-1,:,0:2])
311     # plt.scatter(-local_paths_pixels[-1,:,1], local_paths_pixels[-1,:,0])
312     # plt.show()
313
314
315     valid_opts = list(range(self.num_opts))
316     invalid_opts = []
317     local_paths_lowest_collision_dist = np.ones(self.num_opts) * 50
318
319     print("DONE: Check the points in local_path_pixels for collisions")
320     for opt in range(local_paths_pixels.shape[1]):
321         # for timestep in range(local_paths_pixels.shape[0]):
322
323             rob_center_pixels = local_paths_pixels[:,opt,:]
324
325             # plt.scatter(-self.map_xy[0], self.map_xy[1])
326             # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
327             # plt.show()
328             # print(rob_center_pixels)
329             # plt.scatter(-self.map_xy[0], self.map_xy[1])
330             # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
331             # plt.show()
332             # print(np.shape(rob_center_pixels))
333
334             if self.checkCollision(rob_center_pixels): # if there is a collision
anywhere along timestep
335                 # print(rob_center_pixels)
336                 # plt.scatter(-self.map_xy[0], self.map_xy[1])
337                 # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0], c='
red')
338                 # plt.show()
339                 # exit()
340                 valid_opts.remove(opt) # we immediately remove the option from the
lists
341
342     t3 = rospy.Time.now()
343     collision_det_time = t3 - t2
344     # remove trajectories that were deemed to have collisions
345     print("DONE: Remove trajectories with collisions!")
346
347     # calculate final cost and choose best option
348     print("DONE: Calculate the final cost and choose the best control option!")
349     # final_cost = np.zeros(self.num_opts)
350     final_cost = np.zeros(len(valid_opts))
351     for i in range(0, len(valid_opts)):
352         # print("checking opts")
353         # print(local_paths[-1,i,:])
354         cur_opt = self.all_opts_scaled[i]
355         final_cost[i] = self.calculate_cost(cur_opt, local_paths[-1,valid_opts[i]
],:])
356
357     if final_cost.size == 0: # hardcoded recovery if all options have collision
358     # if np.count_nonzero(final_cost) == 0:
359         control = [-.1, 0]
360     else:
361         best_opt = valid_opts[final_cost.argmin()]
362         # print("chosen_control")
363         # print(final_cost.argmin())
364         # print(best_opt)
365         # print(self.all_opts[best_opt])
366         # print(local_paths[-1,best_opt,:])
367         control = self.all_opts[best_opt]
368         self.local_path_pub.publish(utils.se2_pose_list_to_path(local_paths[:,

```

```

best_opt], 'map'))

368
369     t4 = rospy.Time.now()
370     cost_calc_time = t4 - t3
371     # send command to robot
372     self.cmd_pub.publish(utils.unicyle_vel_to_twist(control))
373
374     # uncomment out for debugging if necessary
375     # print("Selected control: {control}, Loop time: {time}, Max time: {max_time}
376     #       control=control, time=(rospy.Time.now() - tic).to_sec(), max_time=1/
377     #       CONTROL_RATE))
378     # print("traj rollout time: {}, collision det time: {}, cost calc time: {}".
379     #       format(traj_rollout_time, collision_det_time, cost_calc_time))
380     self.rate.sleep()
381
382 def update_pose(self):
383     # Update numpy poses with current pose using the tf_buffer
384     self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link', rospy
385     .Time(0)).transform
386     self.pose_in_map_np[:] = [self.map_baselink_tf.translation.x, self.
387     map_baselink_tf.translation.y,
388     rotation)[2]]
389     self.pos_in_map_pix = (self.map_origin[:2] + self.pose_in_map_np[:2]) / self.
390     map_resolution
391     self.collision_marker.header.stamp = rospy.Time.now()
392     self.collision_marker.pose = utils.pose_from_se2_pose(self.pose_in_map_np)
393     self.collision_marker_pub.publish(self.collision_marker)
394
395 def check_and_update_goal(self):
396     # iterate the goal if necessary
397     dist_from_goal = np.linalg.norm(self.pose_in_map_np[:2] - self.cur_goal[:2])
398     abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])
399     rot_dist_from_goal = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
400     if dist_from_goal < TRANS_GOAL_TOL and rot_dist_from_goal < ROT_GOAL_TOL:
401         rospy.loginfo("Goal {goal} at {pose} complete.".format(
402             goal=self.cur_path_index, pose=self.cur_goal))
403         if self.cur_path_index == len(self.path_tuples) - 1:
404             rospy.loginfo("Full path complete in {time}s! Path Follower node
405             shutting down.".format(
406                 time=(rospy.Time.now() - self.path_follow_start_time).to_sec()))
407             rospy.signal_shutdown("Full path complete! Path Follower node shutting
408             down.")
409         else:
410             self.cur_path_index += 1
411             self.cur_goal = np.array(self.path_tuples[self.cur_path_index])
412         else:
413             rospy.logdebug("Goal {goal} at {pose}, trans error: {t_err}, rot error: {
414             r_err}.".format(
415                 goal=self.cur_path_index, pose=self.cur_goal, t_err=dist_from_goal,
416                 r_err=rot_dist_from_goal
417             ))
418
419 def stop_robot_on_shutdown(self):
420     self.cmd_pub.publish(Twist())
421     rospy.loginfo("Published zero vel on shutdown.")
422
423 if __name__ == '__main__':
424     try:
425         rospy.init_node('path_follower', log_level=rospy.DEBUG)
426         pf = PathFollower()
427     except rospy.ROSInterruptException:
428         pass

```

### 5.3 Navigation : l2\_follow\_path\_myhal.py

```

1 #!/usr/bin/env python3
2 from __future__ import division, print_function
3 import os
4
5 import numpy as np

```

```

6 from scipy.linalg import block_diag
7 from scipy.spatial.distance import cityblock
8 from skimage.draw import disk
9 import rospy
10 import tf2_ros
11
12 # msgs
13 from geometry_msgs.msg import TransformStamped, Twist, PoseStamped
14 from nav_msgs.msg import Path, Odometry, OccupancyGrid
15 from visualization_msgs.msg import Marker
16
17 # ros and se2 conversion utils
18 import utils
19
20
21 TRANS_GOAL_TOL = .1 # m, tolerance to consider a goal complete
22 ROT_GOAL_TOL = .3 # rad, tolerance to consider a goal complete
23 TRANS_VEL_OPTS = [0, 0.025, 0.13, 0.26] # m/s, max of real robot is .26
24 ROT_VEL_OPTS = np.linspace(-1.82, 1.82, 11) # rad/s, max of real robot is 1.82
25 CONTROL_RATE = 5 # Hz, how frequently control signals are sent
26 CONTROL_HORIZON = 5 # seconds. if this is set too high and INTEGRATION_DT is too low,
    code will take a long time to run!
27 INTEGRATION_DT = .025 # s, delta t to propagate trajectories forward by
28 COLLISION_RADIUS = 0.225 # m, radius from base_link to use for collisions, min of
    0.2077 based on dimensions of .281 x .306
29 ROT_DIST_MULT = .1 # multiplier to change effect of rotational distance in choosing
    correct control
30 OBS_DIST_MULT = .1 # multiplier to change the effect of low distance to obstacles on a
    path
31 MIN_TRANS_DIST_TO_USE_ROT = TRANS_GOAL_TOL # m, robot has to be within this distance to
    use rot distance in cost
32 PATH_NAME = 'path.npy' # saved path from l2_planning.py, should be in the same
    directory as this file
33
34 # here are some hardcoded paths to use if you want to develop l2_planning and this file
    in parallel
35 # TEMP_HARDCODE_PATH = [[2, 0, 0], [2.75, -1, -np.pi/2], [2.75, -4, -np.pi/2], [2, -4.4,
    np.pi]] # almost collision-free
36 TEMP_HARDCODE_PATH = [[2, -.5, 0], [2.4, -1, -np.pi/2], [2.45, -3.5, -np.pi/2], [1.5,
    -4.4, np.pi]] # some possible collisions
37
38
39 #Map Handling Functions
40 def load_map(filename):
41     import matplotlib.image as mpimg
42     import cv2
43     im = cv2.imread("../maps/" + filename)
44     im = cv2.flip(im, 0)
45     # im = mpimg.imread("../maps/" + filename)
46     if len(im.shape) > 2:
47         im = im[:, :, 0]
48     im_np = np.array(im) #Whitespace is true, black is false
49     im_np = np.logical_not(im_np) #for ros
50     return im_np
51
52 class PathFollower():
53     def __init__(self):
54         # time full path
55         self.path_follow_start_time = rospy.Time.now()
56
57         # use tf2 buffer to access transforms between existing frames in tf tree
58         self.tf_buffer = tf2_ros.Buffer()
59         self.listener = tf2_ros.TransformListener(self.tf_buffer)
60         rospy.sleep(1.0) # time to get buffer running
61
62         # constant transforms
63         self.map_odom_tf = self.tf_buffer.lookup_transform('map', 'odom', rospy.Time(0),
            rospy.Duration(2.0)).transform
64         # print(self.map_odom_tf)
65
66         # subscribers and publishers
67         self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
68         self.global_path_pub = rospy.Publisher('~global_path', Path, queue_size=1, latch

```

```

= True)
69     self.local_path_pub = rospy.Publisher('~local_path', Path, queue_size=1)
70     self.collision_marker_pub = rospy.Publisher('~collision_marker', Marker,
queue_size=1)

71
72     # map
73     # map = rospy.wait_for_message('/map', OccupancyGrid)
74     # self.map_np = np.array(map.data).reshape(map.info.height, map.info.width)
75     # self.map_resolution = round(map.info.resolution, 5)
76     # self.map_origin = -utils.se2_pose_from_pose(map.info.origin) # negative
because of weird way origin is stored
77     # self.map_nonzero_idxes = np.argwhere(self.map_np)
78     map_filename = "myhal.png"
79     occupancy_map = load_map(map_filename)
80     self.map_np = occupancy_map
81     self.map_resolution = 0.05
82     self.map_origin = np.array([ 0.2 , 0.2 ,-0. ])
83     self.map_nonzero_idxes = np.argwhere(self.map_np)
84
85
86     # collisions
87     self.collision_radius_pix = COLLISION_RADIUS / self.map_resolution
88     self.collision_marker = Marker()
89     self.collision_marker.header.frame_id = '/map'
90     self.collision_marker.ns = '/collision_radius'
91     self.collision_marker.id = 0
92     self.collision_marker.type = Marker.CYLINDER
93     self.collision_marker.action = Marker.ADD
94     self.collision_marker.scale.x = COLLISION_RADIUS * 2
95     self.collision_marker.scale.y = COLLISION_RADIUS * 2
96     self.collision_marker.scale.z = 1.0
97     self.collision_marker.color.g = 1.0
98     self.collision_marker.color.a = 0.5
99
100    # transforms
101    self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_footprint',
rospy.Time(0), rospy.Duration(2.0))
102    self.pose_in_map_np = np.zeros(3)
103    self.pos_in_map_pix = np.zeros(2)
104    self.update_pose()
105
106    # path variables
107    cur_dir = os.path.dirname(os.path.realpath(__file__))
108
109    # to use the temp hardcoded paths above, switch the comment on the following two
lines
110    self.path_tuples = np.load(os.path.join(cur_dir, 'shortest_path.npy')).T
111    # self.path_tuples = np.array(TEMP_HARDCODE_PATH)
112
113    self.path = utils.se2_pose_list_to_path(self.path_tuples, 'map')
114    self.global_path_pub.publish(self.path)
115
116    # goal
117    self.cur_goal = np.array(self.path_tuples[0])
118    self.cur_path_index = 0
119
120    # trajectory rollout tools
121    # self.all_opts is a Nx2 array with all N possible combinations of the t and v
vels, scaled by integration dt
122    self.all_opts = np.array(np.meshgrid(TRANS_VEL_OPTS, ROT_VEL_OPTS)).T.reshape
(-1, 2)
123
124    # if there is a [0, 0] option, remove it
125    all_zeros_index = (np.abs(self.all_opts) < [0.001, 0.001]).all(axis=1).nonzero()
[0]
126    if all_zeros_index.size > 0:
127        self.all_opts = np.delete(self.all_opts, all_zeros_index, axis=0)
128    self.all_opts_scaled = self.all_opts * INTEGRATION_DT
129
130    self.num_opts = self.all_opts_scaled.shape[0]
131    self.horizon_timesteps = int(np.ceil(CONTROL_HORIZON / INTEGRATION_DT))
132
133    self.rate = rospy.Rate(CONTROL_RATE)

```

```

134     self.prev_ctrl = np.array([0,0])
135
136     rospy.on_shutdown(self.stop_robot_on_shutdown)
137     self.follow_path()
138
139     def trajectory_rollout(self, vel, rot_vel, x_0):
140
141         # Given your chosen velocities determine the trajectory of the robot for your
142         # timestep
143         # The returned trajectory should be a series of points to check for collisions
144
145         # x_0 is expected as x, y, theta
146         # x_k_1 = x_k + B(x_k)u_k*dt
147         B = lambda x_theta : np.array([[np.cos(x_theta), 0],[np.sin(x_theta), 0],[0,
148         1]])
149
150         # initialize output
151         trajectory = np.zeros((3, 1))
152
153         # prepare variables needed for computation
154         u = np.array([[vel],[rot_vel]])
155         dt = INTEGRATION_DT
156         # control is computed from the current pose
157         # in the global reference frame
158         currState = np.reshape(x_0, (3,1))
159         nextState = np.zeros((3,1))
160
161         for i in range(0, self.horizon_timesteps):
162
163             # compute B(x_k)
164             B_k = B(currState[2][0])
165             # compute new state
166             #print(np.shape(B_k))
167             #print(np.shape(u))
168             nextState = currState + np.dot(B_k,u)*dt
169
170             # print("state propagation")
171             # print(currState)
172             # print(nextState)
173
174             # adjust heading to be in [-pi, pi]
175             if nextState[2] > np.pi:
176                 nextState[2] = nextState[2] - 2*np.pi
177             elif nextState[2] < -np.pi:
178                 nextState[2] = nextState[2] + 2*np.pi
179
180             # save current state and progress to the next timestep
181             if i == 0:
182                 trajectory = nextState
183             else:
184                 trajectory = np.hstack((trajectory, nextState))
185
186             currState = nextState
187
188         return np.transpose(trajectory)
189
190     def points_to_robot_circle(self, points):
191         #Convert a series of [x,y] points to robot map footprints for collision
192         #detection
193         #Hint: The disk function is included to help you with this function
194
195         # transform robot coordinates to cell coordinates
196         # cellPos = self.point_to_cell(points)
197         cellPos = points
198
199         # create a base region for the radius specified
200         radRes = int((COLLISION_RADIUS/self.map_resolution) + 1# TODO: check if self map
201         resolution is legit
202         rr, cc = disk((0, 0), radRes)
203         baseCoords = np.vstack((rr,cc), dtype = np.int32)
204
205         # create result place holder
206         tempX = np.zeros((2,1))

```

```

203     pts2Rob = {}
204
205     for i in range(0, cellPos.shape[1]):
206
207         # check for duplicates
208         keyLoc = (cellPos[0][i], cellPos[1][i])
209
210         if keyLoc not in pts2Rob:
211
212             # extract current points
213             tempX[0] = cellPos[0][i]
214             tempX[1] = cellPos[1][i]
215             # map the occupancy region
216             tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
217             # adjust for corner cases
218             tempDiff[0, tempDiff[0] >= self.map_np.shape[1]] = self.map_np.shape[1]
- 1
219             tempDiff[1, tempDiff[1] >= self.map_np.shape[0]] = self.map_np.shape[0]
- 1
220             remCells = np.where(tempDiff < 0)[1]
221             tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
222             # save in dictionary only unique coordinates and entries
223             pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)
224
225         return pts2Rob
226
227     def checkCollision(self, points_T):
228         # points are of shape (2, N) in discrete coordinates
229         points = np.transpose(points_T)
230         # get the robot occupancy map for each set of center points
231         pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy
regions
232         # dictionary with keys being center points, and data being an array of
coordinates of rob occupancy
233
234         # check if pose is collision free
235         for key in pts2RobPoses:
236
237             # get robot occupancy map: (2, M) array
238             mapOcc = pts2RobPoses[key].astype(int) # coordiantes based on set passed in
by "points"
239             # print("occupied cells by key:{}".format(key))
240             # print(mapOcc)
241
242             # plt.scatter(-self.map_xy[0], self.map_xy[1])
243             # plt.scatter(-mapOcc[1,:], mapOcc[0,:])
244             # plt.scatter(-key[1], key[0], s = 100)
245             # plt.scatter(-points[1,0], points[0,0])
246             # plt.show()
247             # exit()
248
249             # check if any of the environment occupancy map cells occupied by the robot
contains a value of 0 and report collision as True
250             crash = np.any(self.map_np[mapOcc[1, :], mapOcc[0, :]] == 100) # TODO:
ensure this is how map_np works
251             if crash:
252                 return True
253
254             return False
255
256     def calculate_cost(self, cand_opt, end_pt):
257         # 2 preferences: low dist from goal, low change from previous ctrl
258         trans_factor = 16
259         rot_factor = 0
260         diff_rot_factor = 0
261
262         # difference in goal poses penalized
263         curr_goal = self.cur_goal
264         pose_diff = np.abs(curr_goal - end_pt)
265         loc_cost = trans_factor*(pose_diff[0] + pose_diff[1])
266         rot_cost = rot_factor*pose_diff[2] / loc_cost
267
268         # change in control penalzie

```

```

269     curr_trans_opt = cand_opt[0]
270     last_trans_opt = self.prev_ctrl[0]
271     curr_rot_opt = cand_opt[1]
272     last_rot_opt = self.prev_ctrl[1]
273     ctrl_chg_cost = np.abs(curr_trans_opt - last_trans_opt) + diff_rot_factor*np.abs
274     (curr_rot_opt - last_rot_opt)
275
276     return loc_cost + rot_cost + ctrl_chg_cost # TODO: maybe implement a saturating
277     cost for rotational error
278
279 def follow_path(self):
280     while not rospy.is_shutdown():
281         # timing for debugging...loop time should be less than 1/CONTROL_RATE
282         tic = rospy.Time.now()
283
284         self.update_pose()
285         self.check_and_update_goal()
286
287         t1 = rospy.Time.now()
288         # start trajectory rollout algorithm
289         local_paths = np.zeros([self.horizon_timesteps + 1, self.num_opts, 3])
290         local_paths[0] = np.atleast_2d(self.pose_in_map_np).repeat(self.num_opts,
291         axis=0)
292
293         # propagate trajectory forward, assuming perfect control of velocity and no
294         dynamic effects
295
296         for i, opts in enumerate(self.all_opts):
297             trans_vel = opts[0]
298             rot_vel = opts[1]
299             pred_traj = self.trajectory_rollout(trans_vel, rot_vel, self.
300             pose_in_map_np)
301             local_paths[1:,i,:] = pred_traj #assign complete trajectory one at a
302             time
303             # print(local_paths[:,0,:])
304             # print("start")
305             # print(local_paths[0,:,:])
306             # print("end pts")
307             # print(local_paths[-1,:,:])
308
309             t2 = rospy.Time.now()
310             traj_rollout_time = t2 - t1
311
312             # check all trajectory points for collisions
313             # first find the closest collision point in the map to each local path point
314             local_paths_pixels = np rint((self.map_origin[:2] + local_paths[:, :, :2]) /
315             self.map_resolution).astype(int) #round to nearest int as index
316             # print("local path shape")
317             # print(np.shape(local_paths_pixels))
318             # plt.scatter(-self.map_xy[0], self.map_xy[1])
319             # plt.scatter(-local_paths_pixels[0,:,1], local_paths_pixels[0,:,0])
320             # print(-local_paths_pixels[0,0,0:2])
321             #
322             # print(-local_paths_pixels[-1,:,0:2])
323             # plt.scatter(-local_paths_pixels[-1,:,1], local_paths_pixels[-1,:,0])
324             # plt.show()
325
326             valid_opts = list(range(self.num_opts))
327             invalid_opts = []
328             local_paths_lowest_collision_dist = np.ones(self.num_opts) * 50
329
330             for opt in range(local_paths_pixels.shape[1]):
331                 # for timestep in range(local_paths_pixels.shape[0]):
332
333                 rob_center_pixels = local_paths_pixels[:,opt,:]
334
335                 # plt.scatter(-self.map_xy[0], self.map_xy[1])
336                 # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
337                 # plt.show()
338                 # print(rob_center_pixels)
339                 # plt.scatter(-self.map_xy[0], self.map_xy[1])
340                 # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])

```

```

335         # plt.show()
336         # print(np.shape(rob_center_pixels))
337
338         if self.checkCollision(rob_center_pixels): # if there is a collision
anywhere along timestep
339             # print(rob_center_pixels)
340             # plt.scatter(-self.map_xy[0], self.map_xy[1])
341             # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0], c='
red')
342             # plt.show()
343             # exit()
344             valid_opts.remove(opt) # we immediately remove the option from the
lists
345         t3 = rospy.Time.now()
346         # collision_det_time = t3 - t2
347         # remove trajectories that were deemed to have collisions
348
349         # calculate final cost and choose best option
350         # final_cost = np.zeros(self.num_opts)
351         final_cost = np.zeros(len(valid_opts))
352         for i in range(0, len(valid_opts)):
353             # print("checking opts")
354             # print(local_paths[-1,i,:])
355             cur_opt = self.all_opts_scaled[i]
356             final_cost[i] = self.calculate_cost(cur_opt, local_paths[-1,valid_opts[i
],:])
357
358         if final_cost.size == 0: # hardcoded recovery if all options have collision
359         # if np.count_nonzero(final_cost) == 0:
360             control = [-.1, 0]
361         else:
362             best_opt = valid_opts[final_cost.argmin()]
363             # print("chosen_control")
364             # print(final_cost.argmin())
365             # print(best_opt)
366             # print(self.all_opts[best_opt])
367             # print(local_paths[-1,best_opt,:])
368             control = self.all_opts[best_opt]
369             self.local_path_pub.publish(utils.se2_pose_list_to_path(local_paths[:,
best_opt], 'map'))
370
371             # t4 = rospy.Time.now()
372             # cost_calc_time = t4 - t3
373             # send command to robot
374             self.cmd_pub.publish(utils.unicycle_vel_to_twist(control))
375
376             # uncomment out for debugging if necessary
377             # print("Selected control: {control}, Loop time: {time}, Max time: {max_time
}").format(
378                 # control=control, time=(rospy.Time.now() - tic).to_sec(), max_time=1/
CONTROL_RATE))
379             # print("traj rollout time: {}, collision det time: {}, cost calc time: {}".
format(traj_rollout_time, collision_det_time, cost_calc_time))
380             self.rate.sleep()
381
382         def update_pose(self):
383             # Update numpy poses with current pose using the tf_buffer
384             self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_footprint',
rospy.Time(0)).transform
385             self.pose_in_map_np[:] = [self.map_baselink_tf.translation.x, self.
map_baselink_tf.translation.y,
386                                     utils.euler_from_ros_quat(self.map_baselink_tf.
rotation)[2]]
387             self.pos_in_map_pix = (self.map_origin[:2] + self.pose_in_map_np[:2]) / self.
map_resolution
388             self.collision_marker.header.stamp = rospy.Time.now()
389             self.collision_marker.pose = utils.pose_from_se2_pose(self.pose_in_map_np)
390             self.collision_marker_pub.publish(self.collision_marker)
391
392         def check_and_update_goal(self):
393             # iterate the goal if necessary
394             dist_from_goal = np.linalg.norm(self.pose_in_map_np[:2] - self.cur_goal[:2])
395             abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])

```



```

396     rot_dist_from_goal = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
397     if dist_from_goal < TRANS_GOAL_TOL and rot_dist_from_goal < ROT_GOAL_TOL:
398         rospy.loginfo("Goal {goal} at {pose} complete.".format(
399             goal=self.cur_path_index, pose=self.cur_goal))
400         if self.cur_path_index == len(self.path_tuples) - 1:
401             rospy.loginfo("Full path complete in {time}s! Path Follower node
shutting down.".format(
402                 time=(rospy.Time.now() - self.path_follow_start_time).to_sec()))
403             rospy.signal_shutdown("Full path complete! Path Follower node shutting
down.")
404         else:
405             self.cur_path_index += 1
406             self.cur_goal = np.array(self.path_tuples[self.cur_path_index])
407         else:
408             rospy.logdebug("Goal {goal} at {pose}, trans error: {t_err}, rot error: {
r_err}.".format(
409                 goal=self.cur_path_index, pose=self.cur_goal, t_err=dist_from_goal,
r_err=rot_dist_from_goal
410             ))
411
412     def stop_robot_on_shutdown(self):
413         self.cmd_pub.publish(Twist())
414         rospy.loginfo("Published zero vel on shutdown.")
415
416
417 if __name__ == '__main__':
418     try:
419         rospy.init_node('path_follower', log_level=rospy.DEBUG)
420         pf = PathFollower()
421     except rospy.ROSInterruptException:
422         pass

```